GLISP: AN EFFICIENT, ENGLISH-LIKE PROGRAMMING LANGUAGE

Gordon S. Novak Jr.
Computer Sciences Department
University of Texas at Austin
Austin, Texas 78712

1.0 INTRODUCTION

My earlier research on computer understanding of physics problems stated in English [1,2] has convinced me that English is best viewed as a programming language. That is, an English sentence does not contain the message to be transmitted to the reader, but rather is a program which provides the minimum information necessary for the reader to construct the message from what the reader already knows. In the case of physics problems, the size of the model constructed by the ISAAC program is some 30 times as large as the size of the English sentences which specified the construction of that model. Woods [3] has suggested that the need to communicate complex concepts over a bandwidth-limited serial channel (speech) was the driving force behind the evolution of natural language abilities in humans.

Study of the ways in which English permits compact expression of complex ideas reveals several features which would be useful if incorporated into programming languages. The reader of an English text maintains a current context, or focus [4], which can be used to understand definite references to objects or features of objects which are not specified completely, but are closely related to objects in the current context. For example, consider the following sample of text:

> Last night I went to Scholz's for a beer. The bartender asked for a ride home, since his car was disabled. Somebody had let the air out of the tires.

A person reading this passage can easily understand a definite reference such as "the air", which means "the air which was contained in the tires which are part of the car which is owned by the person who works as a bartender at the bar named Scholz's". The reader has made these connections while reading the story by using world knowledge and by maintaining a current context relative to which definite references to previously unmentioned objects and features can be understood; each reference to an object or feature causes it to be brought into the context, thus enabling further references relative to it. The ability of the reader to infer the connection between a definite reference in a sentence and a "closely related" object in the current context permits the compact specification of complex relationships among objects.

Another valuable feature of English is that it provides a standard interface for communicating information. The writer and the reader may have very different internal representations for certain objects, but they both have procedures for translation between their internal representations and corresponding English descriptions. A related feature is provided by "object-oriented" programming in the SMALLTALK language [5], which is based on the idea of Objects which communicate by exchanging Messages. In most programming languages, object representations are merely storage locations; the nature of the representation is represented implicitly in the programs which manipulate the storage locations. In SMALLTALK, the internal structure of objects is hidden, and programs cannot manipulate the internal structures directly; instead, programs query and change values in the objects by sending them messages, e.g., "what is your X?" or "set your X to the value y". Only the object itself knows whether it actually has an X, or whether its X is a consequence of other values. In addition, the object can act to maintain its own internal consistency; for example, changing the size of an object may require that the object change the size of its picture on a display screen. Unfortunately, SMALLTALK has been implemented on special hardware, and has been unavailable to most researchers.

2.0 NEED FOR ENGLISH-LIKE PROGRAMMING LANGUAGES

English-like programming provides two features which are needed by workers in Cognitive Science and Artificial Intelligence and which are not provided by most existing programming languages: brevity of expression and ease of changing representations. In fact, these two are intertwined: the more detail one has provided about how to perform an action on an object, the more code one will have to change if the basic structure of the object is to be changed. Most existing programming languages implicitly specify the structures of objects within the code. For example, in either PASCAL or CLISP [6], referencing

a field of a record structure requires that both the record and a complete path from the record to the desired field be specified in the code; if the record structure is to be changed, all the code which references such records will often have to be changed also. In a large system, such changes are so difficult that significant changes to data structures are seldom possible once a large body of code exists.

3.0 GLISP

GLISP is a LISP-based language which permits English-like programs containing definite references. GLISP is implemented by a compiler which compiles GLISP programs into LISP relative to a knowledge base which is separate from the programs; the resulting LISP code can be further compiled to machine language by the LISP compiler. In GLISP, the execution of a program causes an implicit context of computation to be constructed, just as an English conversation causes an implicit conversational context to be constructed in the minds of the conversants. The context is computed at compile time, using flow analysis, from the previous context, which includes Structure Descriptions of previously mentioned objects. Definite references to features of objects which are currently in context are permitted; these cause the newly referenced objects to be added to the context, allowing further references relative to them.

The initial context within a GLISP function consists of the arguments of the function, its PROG variables, and any declared global variables. The context contains, for each variable, its variable name, reference name, and Structure Description. When a definite reference is encountered within a GLISP program, the compiler determines whether the reference names such a variable or names a substructure or feature of some variable which is in context. If a substructure or feature is referenced, the compiler determines how to get it from the original structure; the resulting code replaces the definite reference in the compiled version of the program. In addition to producing code to get the feature from the starting structure, the compiler also determines the Structure Description of the result. The new item and its Structure Description are added to the context, thus enabling further definite references relative to it. When the compilation is finished, the context structures disappear; the compiled code contains only the LISP code necessary to perform the specified actions. Thus, the code produced by GLISP is relatively efficient; the user of GLISP must pay for compilation, but does not incur a runtime penalty. The GLISP compiler runs incrementally, so that functions are compiled automatically the first time they are called.

The following example illustrates some of the features of GLISP. Suppose that a wicked witch curses a grandmother by decreeing that each of her calico cats shall age by five years. The code to accomplish this can be written in GLISP as follows:

```
(CURSE (GLAMBDA ( (A GRANDMOTHER) ) (PROG ()
   (FOR EACH CAT WITH COLOR = 'CALICO
        DO  AGE ←+ 5)   )))
```

The GLAMBDA indicates that this is a GLISP function, and causes the GLISP compiler to be called when the function is first interpreted (using the LAMBDATRAN feature of INTERLISP [6]). Since GLISP maintains a context and permits definite reference, it is often unnecessary to give names to variables; thus, we need only declare the type of the argument, (A GRANDMOTHER). Since a GRANDMOTHER is in context, the compiler can determine how to access her CATs and how to generate an appropriate loop to examine each of them. Within the loop, of course, the current CAT is in context, allowing definite reference to its features. The compiler generates the appropriate kind of test to compare the COLOR of the CAT against the constant 'CALICO; if needed, the constant and the operator could be coerced into the appropriate forms. For example, 'CALICO might have several possible meanings; in the context of the COLOR of a CAT, it could be coerced to the unique constant 'CALICO-CAT-COLOR. If the test is satisfied, the AGE of the CAT is increased by 5; the operator ←+, which specifies appending when applied to lists, is interpreted as addition when applied to numbers.

In the GLISP program, we have implied that certain objects have certain features, e.g., that a CAT has a COLOR, but we have said nothing about how to get or replace the COLOR of a CAT, or about what type of entity the COLOR actually is. This information is held separately in the knowledge base of Structure Descriptions and other information relative to which the program is compiled. This makes possible significant changes to data structures with no changes to the code -- a goal long sought in high-level languages, but one which has been largely unrealized for structures involving pointers. GLISP can be viewed as similar to SMALLTALK in the sense that a program does not

specify directly how to manipulate objects. Instead of sending a message to an object, we can think of the GLISP compiler as generating the code to do what the object would do if it received such a message. This provides some of the flexibility of SMALLTALK with high runtime efficiency.

The GLISP compiler allows GLISP expressions and ordinary LISP to be mixed; the user can use as much or as little GLISP as desired. A Structure Description language is provided for the common LISP data structures, and the compiler automatically generates code to access such structures. In addition, the compiler provides a clean interface to one or more representation languages; the user can use both ordinary LISP structures and units in his favorite representation language, accessing both in a transparent manner. A more compact, CLISP-like syntax for GLISP expressions is provided in addition to the English-like syntax. The GLISP compiler, accessing both LISP structures and our GIRL representation language [7], is currently running.

4.0 ACKNOWLEDGMENT

This research was supported by NSF Award No. SED-7912803 in the Joint National Institute of Education - National Science Foundation Program of Research on Cognitive Processes and the Structure of Knowledge in Science and Mathematics.

5.0 REFERENCES

1. Novak, G., "Computer Understanding of Physics Problems Stated in Natural Language", American Journal of Computational Linguistics, Microfiche 53, 1976.

2. Novak, G., "Representations of Knowledge in a Program for Solving Physics Problems", Proc. 5th IJCAI, Cambridge, Mass., 1977, pp. 286-291.

3. Woods, W. A., Symposium on Formal Semantics and Natural Language Processing, University of Texas at Austin, March, 1979.

4. Grosz, B., "The Representation and Use of Focus in Dialogue Understanding", Ph.D. thesis, University of California, Berkeley, 1977. Also Technical Note No. 151, SRI International, Menlo Park, California.

5. Ingalls, D., "the Smalltalk-76 Programming System: Design and Implementation", 5th ACM Symposium on Principles of Programming Languages, Tucson, Arizona, January 1978.

6. Teitelman, W., "INTERLISP Reference Manual", Xerox Palo Alto Research Center, 1978.

7. Novak, G., "GIRL and GLISP: An Efficient Representation Language", submitted to IJCAI-81.